Lab 03: Dictionaries, tuples, external files and password cracking

This lab introduces the Python dictionary and tuple data types. Working with scripts is extended through the script arguments list object. You will also get user input and read data from external data files.

1. Python Data Types – Dictionary Object

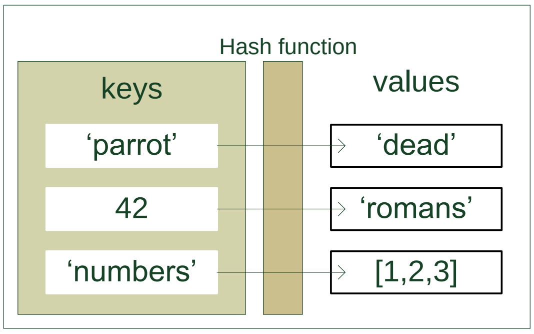
The **Python Dictionary** object is another collection object. Like lists it is mutable, and values can be added and removed.

Python Lists are ***ordered*** by the offset value, and can be sorted and reordered. Python Dictionaries are ***unordered***, though the insertion order is maintained since Python 3.7. Unlike the List, which is indexed with the offset integer, a Dictionary is indexed with a unique key – which can be a string or other basic type. Dictionaries can be thought of as a set of ***key: value* pairs**.

Dictionaries can contain any type of object, and each object in the dictionary can be accessed using the key. Dictionaries in Python are similar to *hash tables/maps* in other languages, and allow direct access to the *value* associated with the *key,* without having to iterate through a collection. When using large collections hash tables can give significant speed advantages over indexed collections.

To associate an object to a key the following format is used:

<key>:<value> A dictionary is composed of key/value pairs such as the following:



Creating a dictionary object, and adding several key:value pairs of literals to it can be done like this:

```
>>> dic = {'parrot': 'dead', 42: 'romans', 'numbers':[1,2,3]}
```

From the IDLE shell window, create the Python dictionary above.

When we want to get a value from the Dictionary, we pass a key, and the associated object is returned (similar to using the index for a list):

```
>>> dic['parrot']
'dead'
```

Q: What is the object returned for dic['numbers']?

As for lists, **any kind of objects** can be used as a value, **however only simple objects can be used as keys**.

The previous example showed how to add objects as we create a dictionary. Another common way is to create an empty dictionary, and add elements to it. This can be done like so:

```
>>> dic = {}
>>> dic['parrot'] = 'dead'
>>> dic[42] = 'romans'
>>> dic['numbers'] = [1,2,3]
```

Again, to access a single value, use the [] operator passing the key (note 42 is the key value and not an index offset):

```
>>> dic[42]                >>>dic['parrot']
'romans'                   'dead'
```

To check if a value is in a Dictionary, the **in** operator can be used (as with Strings and Lists):

```
>>> 'parrot' in dic
True
```

Q: What is the response to the following command

```
    >>> 'dead' in dic
```

Q: Why?

Trying to access a single value which is not in Dictionary gives error:

```
>>> dic['not_in']
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    dic['not_in']
KeyError: 'not_in'
```

To avoid the error, you could use an if statement:

```
>>> if 'parrot' in dic: print (dic['parrot'])
'dead'
```

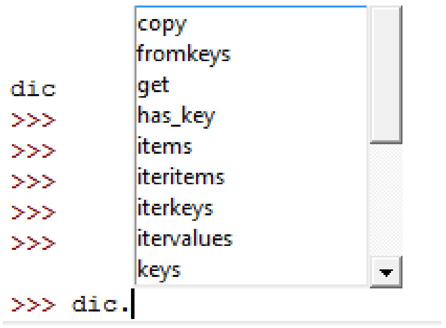
But the .get() method is much more elegant:

Q: What are the results of the following?

```
>>> dic.get('parrot')
>>> dic.get('dead')
```

Q: Describe in your own words what the .get() method does?

In addition to .get(), there are many more dict object methods that provide useful functionality specific to dictionaries. Use `dict.`<CTRL+SPACE> key combination to check the available methods:



```
dic
>>>
>>>
>>>
>>>
>>>
>>> dic.
```

To remove items from a dictionary one by one, returning the values, the **`.pop()`** method can be used:

```
>>> dic['extra'] = 'doomed'
>>> print dic
{42: 'romans', 'last': 3, 'numbers': [1, 2, 3], 'parrot': 'dead', 'extra': 'doom
ed'}
>>>
>>> dic.pop('extra')
'doomed'
>>>
>>> print dic
{42: 'romans', 'last': 3, 'numbers': [1, 2, 3], 'parrot': 'dead'}
>>>
```

To get the entire Dictionary in a more usable format, the **`items()`** method can be used:

```
>>> dic.items()
```

Q: with dic.items(), what type of data structure is output for each key:value pair?

Similarly, to get the keys only, the **`keys()`** method can be used:

```
>>> dic.keys()
```

Q: In what data structure are the keys returned?

Q: Which method outputs only the values, and in what data structure?

(try `dic.`<CTRL+SPACE>)

To check how many items are in the Dictionary, again the BIF `len()` can be used (as with Lists and Strings):

```
>>> len(dic)
```

Q: How many items in dic?

Q: Why not 6?

If you ever need to make a copy of an entire Dictionary, the copy() method can be used:

```
>>> dic2 = dic.copy()
>>> dic2['extra']=2
>>> print(dic)
{42: 'romans', 'last': 3, 'numbers': [1, 2, 3], 'parrot': 'dead'}
>>> print(dic2)
{42: 'romans', 'last': 3, 'numbers': [1, 2, 3], 'parrot': 'dead',
'extra': 2}
```

To remove the entire dictionary the **del** operator is used:

```
>>> del dic2
>>> print (dic2)

Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(dic2)
NameError: name 'dic2' is not defined
```

2. Iteration of Dictionaries

If you need to perform tasks on each item of a Dictionary, you can use a for loop, using the built in Python **`for in`** construct (same as with Lists and strings).

By default the loop iterates over the keys of the Dictionary, and you have to access the value using the key. Try the following:

```
>>> print(dic)
{'parrot': 'dead', 42: 'romans', 'numbers': [1, 2, 3], 'last': 3}
>>> for key in dic:
        print(f'{key} -> {dic[key]}')


parrot -> dead
42 -> romans
numbers -> [1, 2, 3]
last -> 3
```

This is the same as iterating over the list of keys returned by the **.keys()** method. Try:

```
>>> for key in dic.keys():
        print(f'{key} -> {dic[key]}')
```

Q: Is the output the same?

The previous method <u>is not the most efficient</u> to access the value of a dictionary as the dictionary is read twice for each iteration: 1. to get the key, 2. to get the value associated with the key.

It is better to use the **`items()`** method which needs to access the dictionary only once for each iteration, returning a (key, value) tuple. We can then assign to two variables and use them. Try:

```
>>> for key,value in dic.items():
        print(f'{key} -> {value}')
```

Q: Did you notice a performance improvement?

(ok you won't until using very large dictionaries, but it is still good practice to use this method)

3. zip() – creating a dictionary from two lists

Sometimes you may have two separate lists of related items which would make more sense as a dictionary.

For example:

```
english = ['good morning','good evening','see you soon','thank you']
french = ['bonjour', 'bonsoir', 'à bientôt', 'merci']
```

Create these two lists, then create a dictionary from them using the zip function:

```
dict1=dict(zip(english, french))
```

Print dict1 to check what you got.

Q: Which list is used for the keys in dict1? Which list for the values?

Q: What does the dict() function do? What happens if you forget it?

We can now translate easily from English into French using our dictionary, for example:

```
>>> dict1.get('see you soon')
'à bientôt'
```

Q: what is the benefit of using the .get() function here?

Q: How can we use dict1 to translate from French to English?

Because the French phrases are the values of the dictionary, they cannot be used easily as the lookup. Therefore to translate from French to English, it would be better to have a dictionary where the French phrases are the keys. We could create such a dict from the original two lists, giving french as the first argument of the zip function instead. However, imagine you have only the dictionary dict1.

There must be a way to "turn it round", i.e. turn the keys into the values and vice versa. There is no specific object method built-in for this.

4. Dictionary comprehensions **important**

You could use a dictionary iteration to "turn round", but because you are effectively creating a new dictionary, a **dict comprehension** will be more efficient and more elegant. Try:

```
>>> dict2 = {fr:en for (en,fr) in dict1.items()}
>>> dict2
{'bonjour': 'good morning', 'bonsoir': 'good evening', 'à bientôt':
'see you soon', 'merci': 'thank you'}
```

Q: Explain in your own words how this dict comprehension works.

5. External Data - User Input

Python provides a BIF **input()** to get user input from the command line.

To get the name of the user and print it out you could use this code:

```
>>> reply = input('Your name? '); print(f'hello {reply}')
Your name? Petra
hello Petra
```

Create a new Python file, called `add_2_ints.py`, which gets 2 numbers from the user and adds them together, and prints the sum to the command line.

Q: What is the line of code to add the two inputs together? (remember to convert to the correct types)

Q: What is line of code to print the sum out?

Test your code by running in the Python shell.

Q: Does the code work?

Your final code should be similar to:

```
reply1 = input('Please enter a number: ')
reply2 = input('Please enter a 2nd number: ')
total = int(reply1) + int(reply2)
print (f'Sum = {total}')
```

6. For and While

If we wanted to sum more than 2 numbers from the user, we would need a loop. So far, we have used **for loops** as we have known how many iterations we are going to make.

If we know that the user was going to enter 5 numbers, we could create a script:

```
reply_list = []
for i in range(5):
    reply_list.append(int(input('Please enter a number: ')))
total = sum(reply_list)
print (f'Sum = {total}')
```

Note the use of the `range()` BIF. It generates a list of numbers. The following generates a range of 5 numbers.

```
>>> range(5)
[0, 1, 2, 3, 4]
```

It can also be used to generate range of numbers, with other start, stop, and increments.

```
>>> range(10, 100, 10)
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

If we did not know beforehand how many numbers the user was going to enter, a different looping technique would be needed.

The **while loop** can be used if the number of iterations in the loop is not known. Create a new script and try the following code to add any number of numbers from the user. Only when the user enters '0' does the while loop exit.

```
# Module:  add_ints.py
# Desc: Get numbers from user and add them together
# Modified:  Oct 2017

reply_list = []
reply = ''

while reply != '0':
    reply = input('Enter a number (0 to finish): ')
    reply_list.append(int(reply))

total = sum(reply_list)
print (f'Sum = {total}')
```

7. External Data – Reading from Files

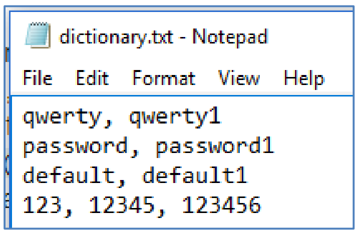
Python has many built in functions and standard modules with file handling and file system interaction functionality. These provide us with OS independent Python functions so the same code will run on Windows or Linux or Apple OSs.

The `open()` **BIF** opens and returns a file object. The built in `file` object has several basic file handling methods, such as readline(), which can be used to process the data from the file after it has been opened.

Create a file **dictionary.txt** with the following text in it:

```
qwerty, qwerty1
password, password1
default, default1
123, 12345, 123456
```

Save it in your normal save location or in the main Python directory. Make a note of the path where you've saved it.



```
qwerty, qwerty1
password, password1
default, default1
123, 12345, 123456
```

From the Python IDLE shell, check the help on the open() BIF using help(open).

Then, use the **open()** BIF to open the file for reading, creating a file object **passwords**

```
>>> passwords = open('dictionary.txt','r')
```

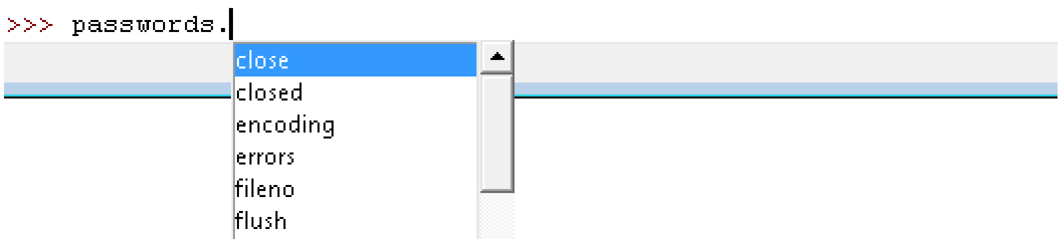
Use type() to check what type of object passwords is:

```
>>> type(passwords)
```

Q: Which type of object is passwords?

Q: What does 'r' mean? (see https://docs.python.org/3.7/library/functions.html#open)

Check the methods available on the File object returned using .CTRL+SPACE:



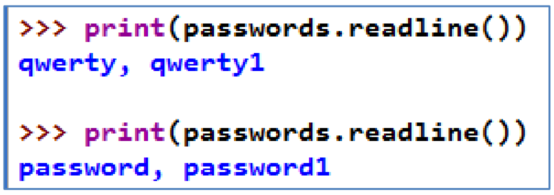
```
>>> passwords.
```

Q: Which methods might be useful to read data from the file object?

Q: Which method reads a single line from the file and returns a string?

Hint: Try typing `passwords.<methodname>(`

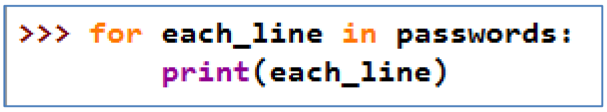
We can use the **readline()** method to read individual lines from the file, and then we can print them to the shell: (remember <ALT+P> to reuse commands from the history)



```
>>> print(passwords.readline())
qwerty, qwerty1

>>> print(passwords.readline())
password, password1
```

8. Iteration of Lines of File

A better way to deal with the lines of the file, is to iterate through the file, using **for in** construct again:



```
>>> for each_line in passwords:
        print(each_line)
```

Q: What is the result?

Q: Why?

You can use the **tell()** method to check which byte the file object is currently pointing at.

```
>>> passwords.tell()
```

To move back to the start of the file, you can **close()** and **open()** again, or use the **seek()** method of the file object:

```
>>> passwords.seek(0)
```

Note: it takes a byte offset as an argument, so by putting 0 we should be back at the beginning of the file – byte 0!

Now, run the previous **for in** statement again to print out all the lines in the file.

Q: Did it print out all the line in the file?

Q: Are there blank lines in between? Why? How might we fix this in the print statement?

When a file has been finished with, you should use the **close()** method on the file object to free up resources.

```
>>> passwords.close()
```

9. Tuple Data Type

For this exercise, use the file **dictionary_variants.txt** which you can download from moodle. Have a look at the file contents. Each line of our data contains a password and one or two variants, separated by commas.

Q: The filename includes "dictionary", but does the file contain a Python Dictionary?

We can use the string method `string.split()` to break up each line of our password file into individual base passwords and password variants, and store in a Tuple. The individual variables from the tuple can then be used separately:

```
>>> passwords = open('dictionary_variants.txt')
>>> for each_line in passwords:
        (passwd, passwd_variant) = each_line.split(',')
        print(f'Password: {passwd}, Variant: {passwd_variant}')
```

Q: What is different about the Tuple syntax and a List syntax?

The **square brackets []** are used to group items in a List, and the standard **round brackets ()** are for Tuples.

Q: What happens if we run the code?

If we run the code, Python returns a ***Runtime Error*** , also known as an ***Exception***:

```
Password: qwerty, Variant:  qwerty1

Password: password, Variant:  password1

Password: default, Variant:  default1

Traceback (most recent call last):
  File "<pyshell#10>", line 2, in <module>
    (passwd, passwd_variant) = each_line.split(',')
ValueError: too many values to unpack (expected 2)
```

Q: Why has an exception been raised?

There is a problem here, in that the last line has more than one variant of the base password. We will cover exception handling next week. For now, let's fix this problem.

Let's check for help with the **split()** method, to see if there is anything we can change:

```
>>> help(each_line.split)
Help on built-in function split:

split(...) method of builtins.str instance
    S.split(sep=None, maxsplit=-1) -> list of strings

    Return a list of the words in S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator and empty strings are
    removed from the result.
```

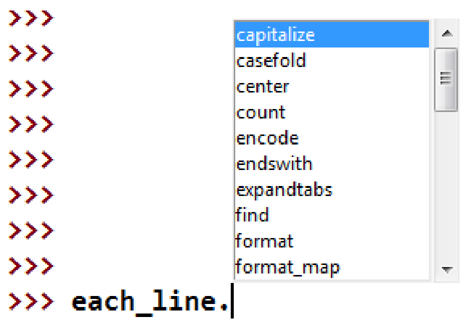
It has a **maxsplit** argument, which should help for now.
Try setting maxsplit=1

Remember to move back to the first byte of the file before running your loop again. Hint: seek()

Q: Did the maxsplit arg get rid of the Error?

```
>>> passwords.seek(0)
0
>>> for each_line in passwords:
        (passwd, passwd_variant) = each_line.split(',',1)
        print(f'Password: {passwd}, Variant: {passwd_variant}')

Password: qwerty, Variant:  qwerty1

Password: password, Variant:  password1

Password: default, Variant:  default1

Password: 123, Variant:  12345, 123456
```

This works, but is not ideal, as we have not unpacked the extra variant for the 123 password when we read in the file. We can check to see if there are any other String object methods which might help. Try .CTRL+SPACE:



```
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>> each_line.
```

The `.count()` method sounds promising. Check for more help on the method itself:

```
>>> help(each_line.count)
Help on built-in function count:

count(...) method of builtins.str instance
    S.count(sub[, start[, end]]) -> int

    Return the number of non-overlapping occurrences of substring sub in
    string S[start:end].  Optional arguments start and end are
    interpreted as in slice notation.
```

Experiment with the method, by reading lines for the file, and using count() to check for comma characters (again remember to go back to the beginning of the file!):

```
>>> line =passwords.readline()
>>> line.count(',')
1
>>> line =passwords.readline()
>>> line.count(',')
1
>>> line =passwords.readline()
>>> line.count(',')
1
>>> line =passwords.readline()
>>> line.count(',')
2
```

Let's incorporate this into the loop.

```
>>> for each_line in passwords:
        (passwd, passwd_variant) = each_line.split(',',each_line.count(','))
        print(f'Password: {passwd}, Variant: {passwd_variant}')
```

Q: Did the .count() method work?

Q: What is the problem?

You should get an Exception, and a Traceback (dump of the call stack) which shows exactly where the run-time error occurred, and the error type; in this case a ***ValueError*** (check last line for the error)

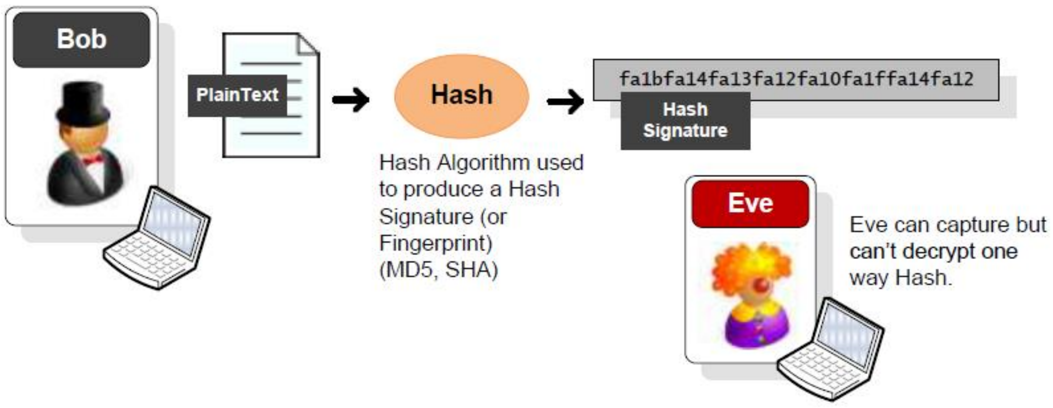
```
Traceback (most recent call last):
  File "<pyshell#41>", line 2, in <module>
    (passwd, passwd_variant) = each_line.split(',',each_line.count(','))
ValueError: too many values to unpack (expected 2)
```

The Tuple only has length 2, so it can handle only one variant on top of the base password.

10. Hash signatures - Python Cyber Security Applications

Hash signatures can be used for authentication of users, files, and even entire digital media.

To generate a hash signature, a hash algorithm is used. In the figure below, Bob creates a hash signature of a file so he can tell at a later date if the file has been changed.


Bob
PlainText
Hash
Hash Algorithm used to produce a Hash Signature (or Fingerprint) (MD5, SHA)
fa1bfa14fa13fa12fa10fa1ffa14fa12
Hash Signature
Eve
Eve can capture but can't decrypt one way Hash.

In the Python Interpreter use the standard library module **hashlib** and the MD5 Hash Algorithm - signature creation function **md5()** - to generate a Hash Signature of a name. Then the **hexdigest()** function can be called to return a printable Hex encoding of the hash.

In Python 3.7, we need to make sure the string's encoding is specified. Utf-8 is now the de facto standard encoding.

```
>>> import hashlib
>>> md5hash = hashlib.md5('Petra'.encode('utf-8'))
>>> print(md5hash.hexdigest())
a2289681db3b897b364d0260f156c397
```

Q: What is the hash signature of your own name?

Q: Is this different if you use lower case only? If yes, is it very different or almost the same?

Q: How is the hash signature being displayed (what encoding)?

Q: What is the length of the MD5 hash signature (number of characters)?

Q: What dictates this length of the hash signature?

Check this against an online MD5 Hash Generator, a such as :

☞ MD5 online Hash Generator: http://www.asecuritysite.com/Encryption/md5

Now, create a hash signature for the txt **'Hashing is simple in Python!!!'**, using code similar to:

```
>>> md5obj=hashlib.md5('Hashing is simple in Python!'.encode('utf-8'))
>>> print(md5obj.hexdigest())
```

Q: What is the length of the hash signature generated?

Q: What would be the length of the MD5 hash signature generated for this entire pdf lab sheet?

SHA1 Hashes

Now, use the Python **help() and dir()** BIFs to find the a SHA1 hash function.

Create a SHA1 hash object for your name and print out the hex encoding of the hash.

Q: What is the hash signature of your own name?

Q: What is the length of the hash signature (number of characters)?

11. Hash Password 'Recovery'

Often passwords are stored as hash digests/signatures. A typical way to 'recover' hashed passwords is to use a dictionary of password words, to hash each word, and compare the each hash against the hash we are trying to crack.

Use the following code as a starting point to create a dictionary-based hash password recovery tool based on some common passwords (NakedSecurity, 2010).

You can copy & paste the code from below or download from moodle to create the initial script.

The script will run but do nothing much. Your task is to complete the three lines that have comments labelled with triple ###.

Script starting point:
dict_crack_start.py **(in Moodle)**

```
# Script:  dict_crack.py
# Description: Cracks password hash using a dictionary attack.
# Author:  Petra L & Rich McF
# Modified: Sept 2018
import sys
import hashlib

# list of passwords
dic = ['123','1234','12345','123456','1234567','12345678',
        'password', 'qwerty','abc','abcd','abc123','111111',
        'monkey','arsenal','letmein','trustno1','dragon',
        'baseball','superman','iloveyou','starwars',
        'montypython','cheese','123123','football','batman']

# create list of corresponding md5 hashes using a list comprehension
hashes = [None for pwd in dic] ### replace None with your formula

# zip dic and hashes to create a dictionary (rainbow table)
rainbow = {} ### replace empty dictionary with your formula

def dict_attack(passwd_hash):
    """Checks password hash against a dictionary of common passwords"""

    print (f'[*] Cracking hash: {passwd_hash}')
    passwd_found = None ### replace None with a lookup using .get() on rainbow

    if passwd_found:
        print (f'[+] Password recovered: {passwd_found}')
    else:
        print (f'[-] Password not recovered')

def main():
    print('[dict_crack] Tests')
    passwd_hash = '4297f44b13955235245b2497399d7a93'
    dict_attack(passwd_hash)

if __name__ == '__main__':
      main()
```

Q: What is the password for the test case in main()?

Now, add tests in main to crack the following password hashes:

Q: What is the password for '5badcaf789d3d1d09794d8f021f40f0e'?

Q: What is the password for '0d107d09f5bbe40cade3de5c71e9e9b7'?

Q: Can you recover the password for ' 5c916794deca0f7c3eeaee426b88f8bd'? (maybe not!)

Now **add code to also check uppercase versions of the passwords**.

Q: What is the password for ' 5c916794deca0f7c3eeaee426b88f8bd'?

Now add code to also check **capitalised first letter versions** of the passwords.

Q: What is the password for ' a67778b3dcc82bfaace0f8bc0061f20e'?

12. Python Script Arguments

If we are creating Python scripts which can be run from the command line, we may want to pass arguments to the script. Python stores these arguments in the **sys.argv** object.

Create a script in the editor, from the IDLE shell (using file>new) and save it as **args.py**. Add the following code. You can copy&paste the code from the link below to create the initial script.

Script starting point: args_startcode.py (in moodle)

```
# Script: Manipulate Script Arguments
# Author: Rich Macf & Petra L
# Modified: Oct 2017
#
import sys

TRACE=True

def print_args(arg_list):
        '''print out the arguments passed in as a list'''
        print (f"[print_args] Args: {arg_list}")

def main():
    print ('[print_args] Tests')
    print_args(sys.argv)

# boilerplate code to call the main() function to begin
# the program if run as a script.
if __name__ == '__main__':
    if TRACE: print ('[T] Module called as script, calling main()')
    main()
else:
    if TRACE: print ('[T] Module imported as library, not calling main')
```

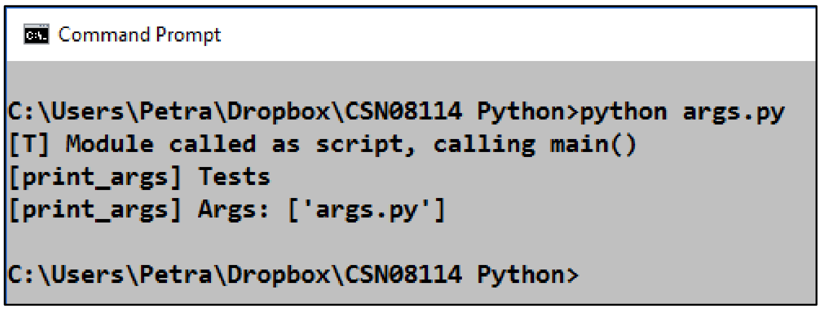
Test from the Python IDLE shell, (1) by importing the module and then (2) by running it as a script from the editor window.

Q: What Python object type is sys.argv?

Q: What is the content of sys.argv?

Q: Do you understand how the boiler plate code is working for each type of test? (if no ask tutor to explain now!)

Test the args.py script from the windows cmd line.



Command Prompt

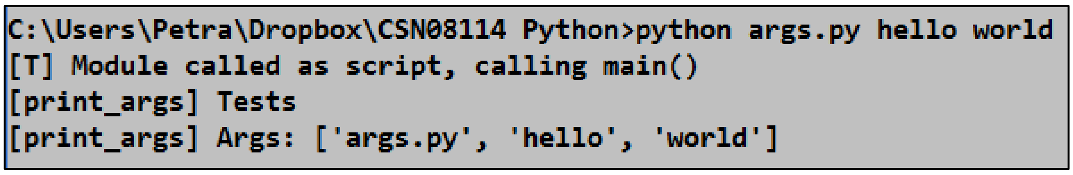
```
C:\Users\Petra\Dropbox\CSN08114 Python>python args.py
[T] Module called as script, calling main()
[print_args] Tests
[print_args] Args: ['args.py']

C:\Users\Petra\Dropbox\CSN08114 Python>
```

Now test again, passing it some arguments such as **hello world:**

```
C:\...\python args.py hello world
```

Q: How many objects are in the sys.argv list now?

The first item in the sys.argv list is the script/module name, and the rest should be the arguments passed in by the user.

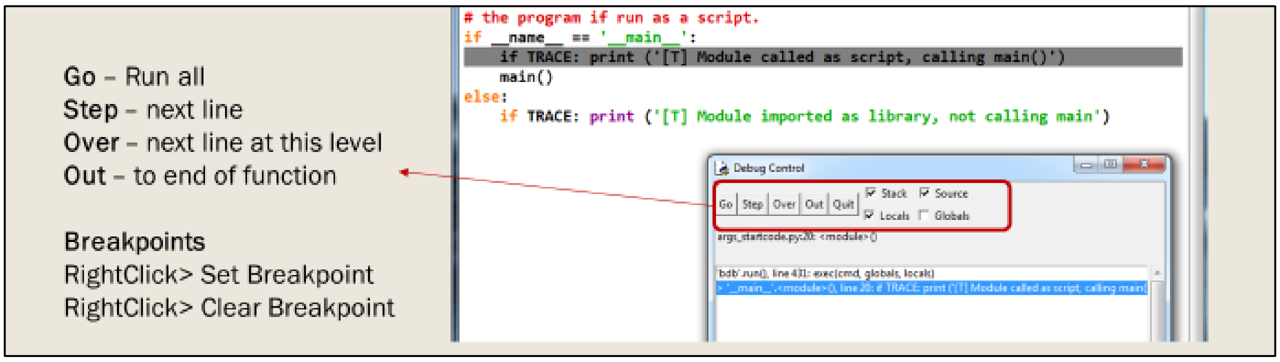


```
C:\Users\Petra\Dropbox\CSN08114 Python>python args.py hello world
[T] Module called as script, calling main()
[print_args] Tests
[print_args] Args: ['args.py', 'hello', 'world']
```

13. IDLE Debugger

Try using the debugger to step through the code of args.py as it is.

From the Python IDLE Shell toggle the Debugger on/off using **Debug > Debugger**, and select the Source checkbox to show the source at runtime.



Go – Run all
Step – next line
Over – next line at this level
Out – to end of function

Breakpoints
RightClick> Set Breakpoint
RightClick> Clear Breakpoint

```
# the program if run as a script.
if __name__ == '__main__':
    if TRACE: print ('[T] Module called as script, calling main()')
    main()
else:
    if TRACE: print ('[T] Module imported as library, not calling main')
```

Debug Control

Go | Step | Over | Out | Quit   [✓ Stack] [✓ Source] [✓ Locals] [ ] Globals

```
args_startcode.py:20: <module>()

'bdb'.run(), line 431: exec(cmd, globals, locals)
> '__main__'.<module>(), line 20: if TRACE: print ('[T] Module called as script, calling main()
```

Python IDLE GUI Debugger

Now from your Module editor window run the module code, and use the **Step/Out** buttons to step through your code/step out of any system code you are not interested in.

Try Setting a breakpoint and running again, using the **Go button** to run the code to the breakpoint, then stepping through the rest.

Toggle the debugger off for now.

14. Adding functionality to the script

Add some code to main(), before the call to print_args(), to check if the user has input any arguments. If not, print a usage message, such as:

```
'Usage: args.py argument1 [argument2] [argument3] ... '
```

(hint: use a BIF to check how may items in list – use lecture slides/ earlier lab exercises as ref). In the same code block as the usage print, exit the code script at that point using:

```
sys.exit(1)
```

Test from the command line with no arguments.

Q: Did the script correctly print out the usage string, and exit?

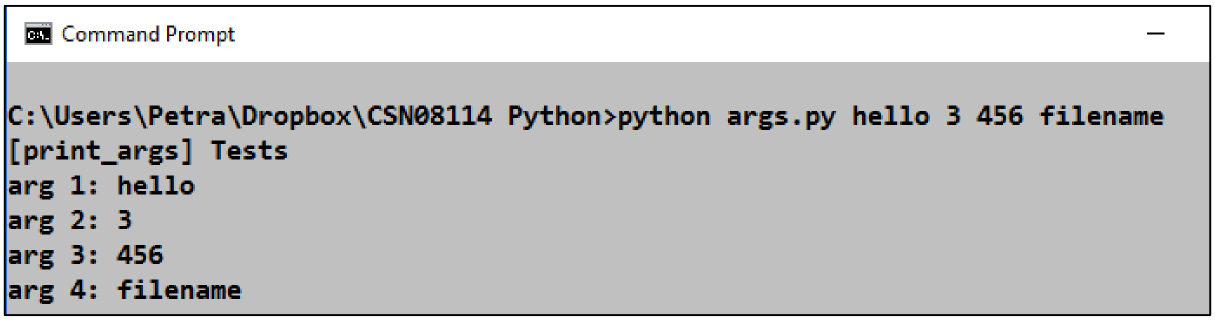
Add some code to main() to call print_args() with only the arguments, and not the name of the function. (hint: the list slice operator will be useful – use lecture slides a reference)

Test from the command line with no arguments.

Q: Did the script correctly print out only the "real" arguments and not the name of the script?

Add some code to the print_args() function to iterate over the argv list and print out each individual argument on a different line. (hint: list iteration – use lecture slides a reference)

Now add code to add output the following: (hint: use string formatting – see lecture 1/lab 1)



```
C:\Users\Petra\Dropbox\CSN08114 Python>python args.py hello 3 456 filename
[print_args] Tests
arg 1: hello
arg 2: 3
arg 3: 456
arg 4: filename
```

15. Testing Scripts with Args from IDLE

Q: If sys.argv is a List object how might we add our own test arguments from <u>within</u> the script?

A test argument can be added to the list just like any other list. In main try:

```
sys.argv.append('argument1')
print_args(sys.argv)
```

And test the script, by running from IDLE using F5 (not the command line)

Q: How might we add multiple test arguments? What code might we use?

Try adding 3 test arguments.

16. Challenge: added features for password cracker

Prerequisite: dict_crack.py password hash recovery script from exercise 11 above.

Use the args.py script from above for inspiration for this exercise!

Challenge A: Add code to the dict_crack script so that the user can run the script from the command line. When running from command line, this should accept as an argument the password hash to be recovered. Add a suitable usage string message too.

Challenge B: Add a second argument so you can specify the hash algorithm to be used, MD5 or SHA1. Add code to your script to recover either an MD5 or SHA1 password hash as specified. Assume MD5 as the default if the hash algorithm isn't specified.

Challenge C: Before carrying out the dictionary attack, we should check that the given password hash is a string which contains only valid hex and is the correct length for an MD5 hash.

Add code to your script to achieve this.

(Hint: iteration is good for checking each char, and the string membership operator may be useful)